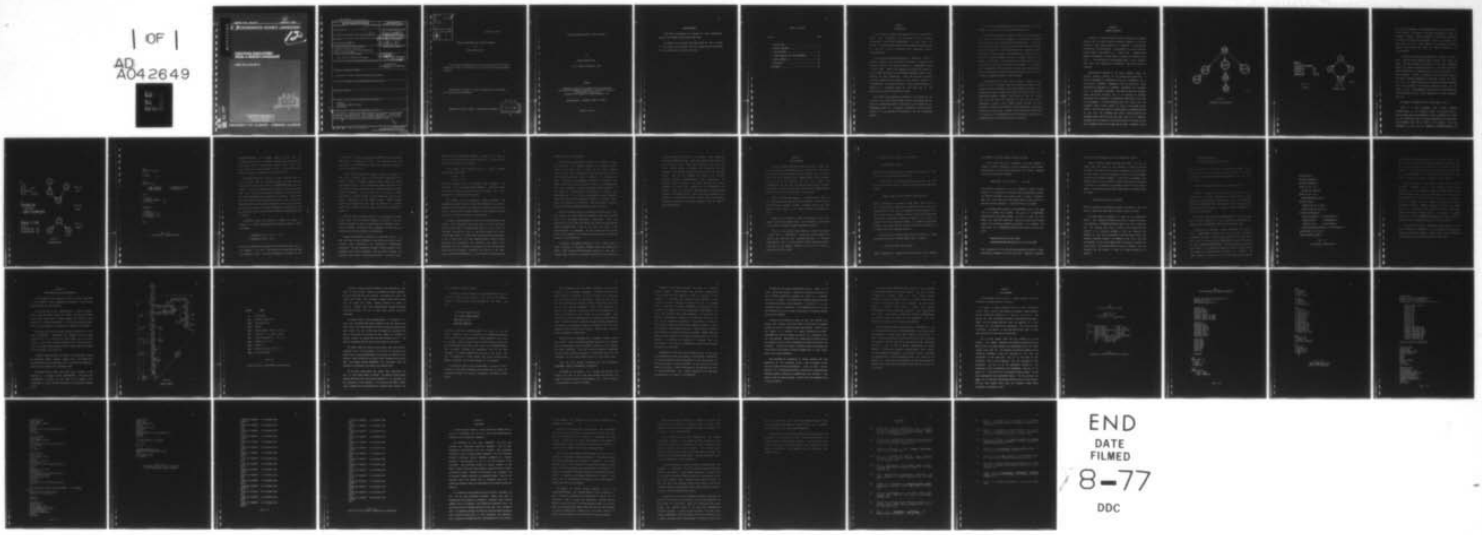
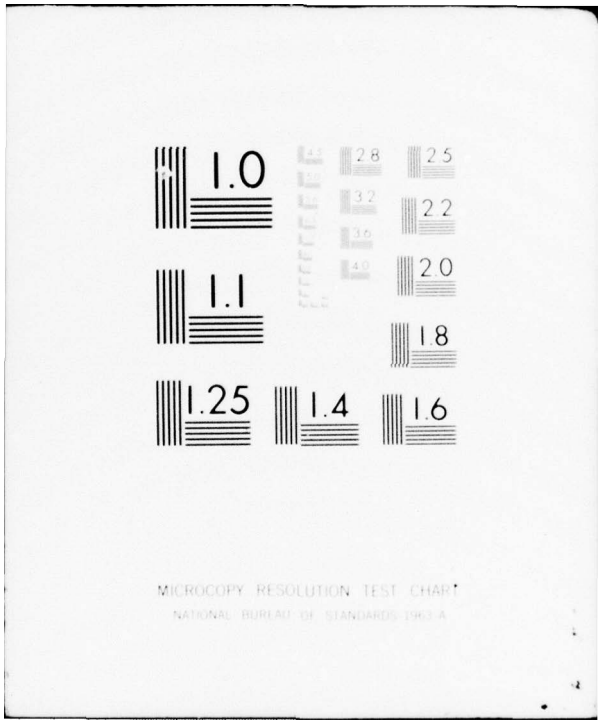
AD-A042 649 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
CHEATING SIMULATORS FROM A DESIGN LANGUAGE.(U)
JUL 77 F M SMITH DAAB07-72-C-0259
UNCLASSIFIED R-773 NL

| OF |
AD A042649



END
DATE FILMED
8-77
DDC


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

REPORT R-773 JULY, 1977 UILU-ENG 77-2220

COORDINATED SCIENCE LABORATORY

12

CHEATING SIMULATORS FROM A DESIGN LANGUAGE

FRANK MYKLAND SMITH

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: CHEATING SIMULATORS FROM A DESIGN LANGUAGE,	5. TYPE OF REPORT & PERIOD COVERED: Technical Report,
	6. PERFORMING ORG. REPORT NUMBER: R-773, UILU-ENG-77-2220
7. AUTHOR(s): Frank Mykland Smith	8. CONTRACT OR GRANT NUMBER(s): DAAB07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Joint Services Electronics Program	12. REPORT DATE: July 1977
	13. NUMBER OF PAGES: 44 (12 50p.)
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Simulator
Asynchronous Digital System
Determinate

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

In an attempt to formalize the design process for large digital systems many researchers have advocated the use of design languages(DL). In this report we present a translator which creates machine simulators from one such DL; this DL is used to describe large asynchronous digital systems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 097700

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

UILU-ENG 77-2220

CHEATING SIMULATORS FROM A DESIGN LANGUAGE

by

Frank Mykland Smith

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

CREATING SIMULATORS FROM A DESIGN LANGUAGE

BY

FRANK MYKLAND SMITH

B.S., Clemson University, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Gernot A. Metze

Urbana, Illinois

ACKNOWLEDGEMENT

This thesis is dedicated to my parents for their wholehearted support and guidance through these many years.

My thanks go to Bill Lahti and Trevor Mudge for their reviews of the thesis and my grammar, and to my thesis advisor Gary Metze for his many helpful criticisms and ideas.

Table of Contents

Chapter 1

INTRODUCTION

In an attempt to formalize the design process for large digital systems many researchers have advocated the use of design languages(DL) [Bar72,Chu72,Dul,Mud75,Met]. In this thesis we present a translator which creates machine simulators from one such DL; this DL is used to describe large asynchronous digital systems[Mud,Pet].

The purpose for building simulators is threefold. First, a software simulator will facilitate the testing of a design at the functional level and will obviate testing of a breadboarded design at this level. Second, the simulator will permit the designer to easily change the design and try different approaches without costly rewiring and time expenditures. Third, the simulator can detect structural problems such as hangups in the control structure or non-determinism of the data structure. These problems might not be detected in a hardwired design for some time after it was operational and would be difficult to locate.

This thesis will describe how simulators are created by the translator and some of the problems that were encountered that are unique to asynchronous systems. This thesis is not intended to be a programming manual or a detailed description of each and every routine. It is an overview of the translator and the translating process.

Let's see how the different phases of the translator are tied together. Then we will look at each phase individually.

First the data structure is read and a data base is built based on a series of declaration statements. In the process of creating the data base, we check for errors and inconsistencies. If the data base is largely correct, we will go to the second phase, in which we analyze the syntax of the input and generate the appropriate code. The syntax is analyzed using a state diagram and a parser, which has some error recovery procedures to patch up simple errors so the parsing can continue. If all errors can be corrected we then go to the last phase, the actual simulation of the design. The simulator that is generated by the translator is passed to the SAIL compiler, and then compiled and executed. With this simulator, the user can use SAIL's debugging package to test the design.

The structure of this thesis is as follows. Chapter 2 will review the DL, discussing some of the limitations imposed on the register transfer statements, some of the problems inherent to an asynchronous digital design, and finally the choice of SAIL as our programming language. Chapter 3 will discuss the idea of a data base, how data types are declared and how the data base is implemented. Chapter 4 will examine syntax analysis and code generation. Chapter 5 will present a design example and a simulator created to test the design. Chapter 6 will conclude the report with a discussion of the methodology used in building the translator and different areas of future growth for the translator.

Chapter 2

GENERAL STRUCTURE

A design is a complex network of control structure (CS) modules forming the CS, which controls register transfers in the data structure. This network consists of a hierarchy of sub-networks organized in a top down manner. A sub-network at a higher level can control several sub-networks at a lower level. Communications between sub-networks go from level to level but not laterally, see Fig 2.1 . This collection of sub-networks forms a tree structure where each node represents a sub-network. Each node is called a "process".

Each process is composed of two basic statement types, the register transfer (reg-trf) or the process call (proc-call). A block of statements is illustrated in Fig 2.2a. Statements 1 and 3 are reg-trf statements. Statements 2 and 4 are proc-calls. These statements are analogous to assignment expressions and procedure calls in programming languages. The order in which the statement types, reg-trf and proc-call, are to be executed is indicated explicitly by the number(s) in parentheses (order-info) to the right of the statement. If order information does not appear then the statement number (order number) is taken to be statement number zero(0). This can most easily be illustrated by a directed graph as in, Fig 2.2b. The ordering must form a partial ordering with statement number zero (0) as the top most node in the ordering. Statement zero is implied to be the entry point of the block, and as such, statement number zero is <u>not</u> used by basic statements except

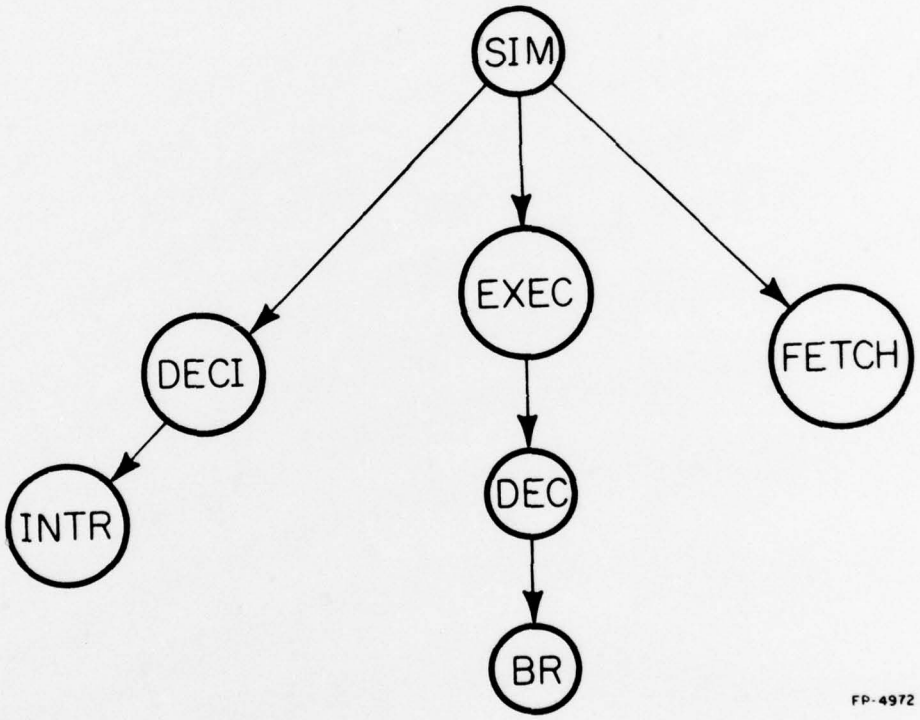


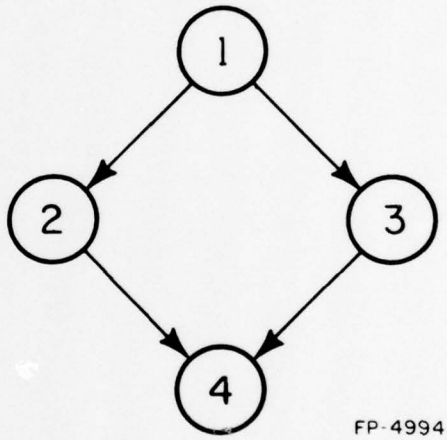
Fig 2.1

Hierarchy of Sub-networks

Block 1
1) Dest ← Source
2) Block 2 (1)
3) D ← S (1)
4) Block 3 (2,3)

Fig 2.2a

A Block



FP-4994

Fig 2.2b

Directed Graph

the control statements (to be discussed later) which have no effect on the ordering. This method of explicitly declaring the order of statement execution allows the designer to declare the concurrency he wishes in executing the sub-network. This is an excellent tool that other DL's don't have, however, it is potentially dangerous because unrestricted use may cause the control structure to be non-determinate.

A collection of these basic statements is generally called a block. There are three different types of blocks: PROC, DPROC, and WPROC. The PROC represents a collection of the basic statements without any control structure,see Fig 2.3a. The DPROC represents a branch depending on its branch variable; this is analogous to a CASE statement. Control is passed to one of the blocks listed in the DPROC or returned immediately to the block calling the DPROC, Fig 2.3b. The WPROC consists of a collection of basic statements but is only initiated if the predicate, controlling entry into the block, is true. Upon completion the block will be reinitiated only if the predicate is true; this is analogous to a WHILE statement, Fig 2.3c.

We present an example using all three blocks in Fig 2.4.

There is a third statement type, control statement (control-stmt), in the translator which is not found in the DL. This statement type was incorporated to aid in debugging the design and in controlling some of the code generated for the simulator. This statement type has no effect on the design. The three statements of this type are CHECKPOINT, OPERATION!CHECKION, and

P

1) A
2) B (1)
3) D ← S
4) A (2,3)

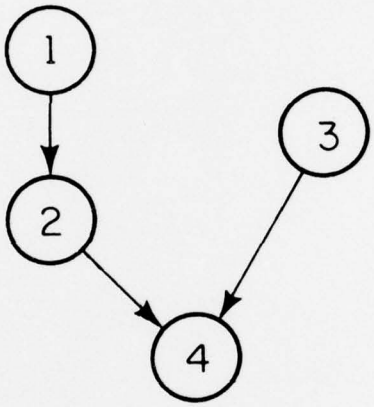


Fig 2.3a

PROC

D

DECODE × AS

00 ⇒ A

NONE ⇒ COMPLETE

Fig 2.3b

DPROC

I

WHILE X=1 DO

1) A
2) D1 ← S1 (1)
3) D2 ← S2 (1)

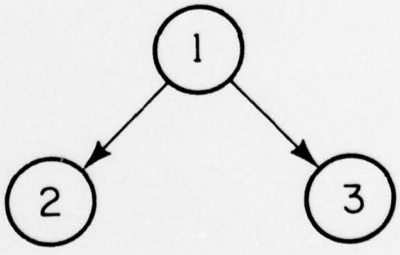


Fig 2.3c

WPROC

FP-4993

Fig 2.3

Example Blocks

```
LOOP
WHILE RUN = ON DO

1) DECI
2) FETCH        (1)


DECI
DECODE INT AS

     1 DOES INTERRUPT          ! INTERRUPT SIGNALED;
     0 DOES COMPLETE           ! NO INTERRUPT;


FETCH

1) MAR<-PC
2) DATA!REG<-MEMORY     (1)
3) PC<-PC +1            (1)


INTERRUPT

1) MAR<-0
2) MEMORY<-PC   (1)
3) MAR<-MAR + 1 (2)
4) PC<-MEMORY   (3)


END
```

Fig 2.4
A collection of connected blocks

OPERATION!CHECK!OFF. The statement number for this type of statement should be zero; otherwise, a mystery warning message will appear which reflects a mistake which actually wasn't made. The control-stmts may appear anywhere within a WPROC or PROC block without affecting the execution or statement ordering.

OPERATION!CHECK!ON will indicate to the translator that it should generate code for detecting overflow and other conditions (zero,positive) after every reg-trf. OPERATION!CHECK!OFF does the opposite, telling the translator not to generate any status setting code. Thus, the user can decrease the simulator's size by judicious use of these statements when checking isn't necessary. The switches take effect when they are "seen" by the parser, and as a result, any code within the same block which precedes the switch is not affected by the switch's action. The order information, in parenthesis, has no significance when used with these switches and consequently will be ignored.

CHECKPOINT allows the designer to output the values of variables at certain key points while executing a WPROC or a PROC. For example,

```
0) CHECKPOINT A,B,C [ 1 to 3]    (4,8)
0) CHECKPOINT D,E,F[ 1 to 5 ]
```

The first statement will generate code to output the values of A, B, and elements 1 to 3 of C. The code will be placed after the code for statements 4 and 8. The second CHECKPOINT is different in that

no order-info is used, which means the CHECKPOINT will be executed immediately on entering the block. The order-info tells the translator to put CHECKPOINT code after every statement appearing in the order information list.

Several design languages use register transfer statements as their basic statement type, but the degree of complexity of the register transfer statements differs greatly among the design languages[Bar75,Dul]. We are limiting the register transfers to be either unary or binary operations, thus preventing complex register transfer statements. We feel that this is closer to a "true" description of a digital system, and more importantly, closer to the actual construction of the digital system. Complex boolean statements are permitted but they must be handled in a special manner when building the data base, which will be discussed in Chapter 3.

There are several problems unique to an asynchronous digital system design and this design language in particular. These are discovering hangups in the CS, insuring that the statements form a partial ordering, serializing the statements for execution ,and insuring that the data structure is determinate.

Hangups are analogous to deadlocks in operating systems. To look for these, we must determine if any direct cycles exist in the directed graph representing the inter-block (INTER!BLOCK) connections. For the intra-block structure we must determine if the statements form a partial ordering. There are several methods for detecting these direct cycles[Joh,Knu,Mud75]; we chose to use a

method called the topological sort[Knu]. We must also be able to serialize the intra-block ordering for execution. This task is also done by the topological sort routine.

As an example, if the statements aren't in a partial ordering the following could occur:

2) A<-B (3)

3) C<-D (2)

which says statement 2 is to be executed after statement 3 but statement 3 is to be executed after statement 2. This would result in a hangup; causing the entire block to be ignored, and a null procedure to be generated for this block.

The biggest and most intractable problem resulting from asynchronous design is that of testing for determinism. A system is determinate if the result of the computation depends uniquely on the initial contents of the storage elements for any execution sequence that does not violate the process's ordering[Cof].

A system is non-determinate if one of the following three cases occurs: read after write, write after read, or write after write. The read after write or the write after read case occurs when for concurrent instructions i and j, i reads from a region and j writes into the same region. In write after write concurrent instructions, i and j write into the same region[Cof,Pet]. Since all these operations are asynchronous and concurrent, the system can't guarantee that the sequence of operations will be the same every time the block is executed. Therefore, the results in the above cases would be non-determinate. We are currently investigating two

different tests for determinism.

The first test, which would be easiest to implement, creates two simulators. One simulator executes all concurrencies that appear in a block from left to right, and the other simulator executes these same concurrencies in the opposite order (right to left), thus reversing the order of execution. By checking the output visually the user determines if there is a difference in the outputs, which would indicate that the design is non-determinate. The visual inspection is one of the method's major drawbacks. If the network is non-determinate the problem is locating this non-determinism. To do this, the user must move output statements around until the group of statements forming a concurrency which caused the problem(s) is pinpointed. Thus, this method is good for quickly detecting non-determinism but not for locating it.

The other possible solution is achieved by forming "read" sets and "write" sets which contain the elements of the right and left part of the assignment expressions respectively. Then the intersection of the "read" and "write" sets where the concurrencies appear is examined[Cof]. If it is not null, the cause of the non-determinism is identified. We have glossed over much of the detail in this approach. The approach is excellent for locating the cause(s) of non-determinism.

In choosing a programming language, we had a choice from a large collection of languages: FORTRAN, ALGOL, SAIL, PASCAL, and SIMULA. After analyzing the programming involved we saw that a language with flexible I/O, character handling, and the ability to

manipulate linked lists would be a necessity. This immediately eliminated FORTRAN and ALGOL. PASCAL was eliminated because its I/O was not flexible enough. SIMULA was eliminated since there was little systems backup if problems were encountered with the language. This left SAIL [Rei] which turned out to be an excellent choice. SAIL has very powerful and flexible I/O, excellent character handling, straightforward record manipulation, and excellent macro facilities. The record structure was used to implement the linked list structure for the topological sort routine. Heavy use was made of the macro facility throughout the translator to parameterize it; most of the code we generate for the simulators is in the form of macros. Macros along with conditional compilation enabled us to tailor the simulator's code and make it much more efficient.

Chapter 3

DATA STRUCTURE

In order to simulate register transfer operations, there must be data elements on which operations can be performed. This raises the problems of how to determine the data elements, how to declare them, and how to represent them. These problems will be discussed in the rest of this chapter. We will also discuss the implementation of the data base and the significance of the data base entries. Lastly, we will look at the procedure INTEGRITY which checks the data base to insure that it is error free.

A data base serves two purposes. It establishes the width of the data paths and the data type of each element. The data base in combination with register transfer statements implies the data structure (DS), which is the entity on which the simulation is based.

A data base is created by a series of declarations which bind object names to data element types. These statements are analogous to declaration statements in most programming languages.

There is a rich variety of data elements, allowing the designer a great deal of flexibility. In the near future, hardware will replace many functions now performed/simulated in software, (specifically, parts of the operating system [Chu72b,Chu77,Smi]). Thus, many of the elements allowed in this DS will become a necessity in future designs.

The most basic data element is the register:

REGISTER Name (width)

where Name is the register's name and "width" is the width of Name in bits, not to exceed some maximum width.

A useful element is the named subfield, which allows the user to name a particular subset of contiguous bits within any other data element.

SUBFIELD Name1 (width) OF Name2 (beginning-point)

Name1 is defined to be a subfield of Name2, where Name1 starts at bit beginning-point in Name2 and is "width" bits wide, the bits are numbered right to left, with the rightmost bit being numbered zero. The subfield idea is extended to several of the other declarations by prefixing the "regular" declaration statement with "subfld" to indicate that this declared element is a subfield of some other element. This allows the user to specify that certain subfields of a larger field have a specific function.

A declaration that generates more than one element is a member of Vector-class, which includes MEMORY, STACK, and QUEUE.

Vector-class Name (size) (width)

Name is defined to be a member of Vector-class with "size" elements

(0 to size-1), with each element "width" bits wide.

To get around some of the limitations that were imposed on register transfer statements, we have a declaration called DEFINE. DEFINE creates a function which allows only the boolean operators and/or the concatenation operator.

```
DEFINE Name = opn1 op1 opn2 .....  opj opnk
```

Thus, Name can be defined to be an arbitrarily complex boolean function or a collection of different fields concatenated together. The operator precedence is the same as SAIL's operator precedence. Name can now be used just as any named register would be used in a register transfer statement. A width for Name is calculated.

An important data element is the MEM!ADDRER, which is the only link to a MEMORY data element. The value of the MEM!ADDRER associated with some memory element is used to index into this memory for read/write operations. There is no other way to access memory, so if some data elements are declared to be MEMORYs then there must be a MEM!ADDRER associated with them or an error will occur.

```
MEM!ADDRER MAR(16) FOR MEM1,MEM2
SUBFLD!MEM!ADDRER MAR!CACHE(8) TO PC(12) FOR CACHE
```

MAR is declared to be 16 bits wide and associated to MEM1 and MEM2, which must be MEMORYs or an error will occur. MAR!CACHE is declared

to be an 8 bit wide subfield of PC and associated to CACHE.

After a register transfer operation takes place, how can the design check the result of the operation or check if an error occurred? This brings up the concept of dependency, for which there are two types. The first dependency is based on the data elements involved in the register transfer. If the object is dependent upon another object of data type STACK or QUEUE then whenever stack/queue is involved in an operation its dependent variable is set to reflect whether the stack/queue overflowed, underflowed, or remained within its limits.

DEP!VAR Name1 (width1) FOR Name2

Name1 is declared to be dependent on Name2; for Name2 to have any effect on Name1 then Name2 must be either a STACK or a QUEUE.

The second type of dependency is based on the operation's result. This dependent variable can be set to indicate the result of an operation, if OPERATION!CHECK!ON switch has been previously set. The switches will have no effect if no variable has been declared to be operation dependent. After the operation two segments of code will be generated to set the correct bits in the operation dependent variable. One segment of code tests and sets for overflow, and the other segment tests for positive, negative, or a zero result. The designer need not use this "feature", but can implement his own design to check for these conditions, if so desired.

OPERATION!CHK Name1(4)

SUBFLD!OPER!CHK Name2(4) TO Name3(2)

Name1 is declared to be 4 bits wide and operation dependent. Name2 is declared to be 4 bits wide and a subfield of Name3. If more than one element is declared to be operation dependent, only the first will be used.

A group of declaration statements is presented in Fig 3.1.

In order to implement the data base, there must be some universal way to create descriptors for each declaration. Three routines are used to create new descriptors. They are DATA!BLOCK, NAMING, and INSERT!NAME!TYPE. These routines set up the descriptors and then call other routines to collect further information to complete the descriptors.

We use two arrays ID!LIST and SYMBOL!TABLE as the data base. They are both indexed by a hash number that is generated by a hashing function (HASH!FUNCT) which uses the object's name.

ID!LIST has the character string representing the object's name. SYMBOL!TABLE's entries hold all the other information concerning the object. The entries are affected by the data type of the object. The first entry in SYMBOL!TABLE is always the object's data type (STACK,REGISTER,SUBFIELD). The second entry is either the hash number of the dependent variable for stacks and queues or the hash number of the field of which the object is a subfield. The third element is either the size of the Vector-class or the bit

```
REGISTER RUN(1)
REGISTER INSTR!WRD(4)
REGISTER PAGE!ADDR(4)
STACK STK (10) (12);
   DEP!VAR A(2) FOR STK;
MEMORY M(1024) (12) ;
   MEM!ADDRER MAR(24) FOR M ;
REGISTER DR(36);
   SUBFIELD DR1(12) TO DR(0) ;
   SUBFIELD DR2(12) TO DR(12);
REGISTER STATUS!WRD(20)
   SUBFIELD POS!NEG(1)   TO STATUS!WRD(17)
   SUBFIELD ZERO(1)      TO STATUS!WRD(16)
   SUBFIELD PC(16)       TO STATUS!WRD(0)
   SUBFLD!OPER!CHK OVFLW!BIT(1) TO STATUS!WRD(18)
DEFINE BOOL=( B OR C) & E;
DEFINE ADDRESS= PAGE&PC;
```

Fig 3.1

A collection of declarations

location where the subfield will start. The fourth entry is the width of the object in bits (fields are numbered right to left with the first bit being zero). The fifth and last entry holds the hash number of the object which acts as a memory address register(MAR) for a specified MEMORY(s). All the entries in the table for any one object are generally not used, allowing for future expansion.

The last procedure to deal with the data base is called INTEGRITY. INTEGRITY's job is to go through the data base and insure that it is self consistent. There are several problems that INTEGRITY searches for. First, it checks that all pointers point to valid elements and that they don't point back on themselves. Second, it checks to see if certain data types meet a minimum requirement for width; dependent variables for STACKs and QUEUEs must be at least two bits wide. Third, it checks if Vector-class elements have their associated variable. For instance, all memory elements must have a MEM!ADDRER associated with them; STACKs and QUEUEs must have a dependent variable for error conditions.

This covers the procedure DATA!BLOCK which oversees all of these activities. DATA!BLOCK is the controlling procedure which contains several other routines. These routines assist DATA!BLOCK in creating the data base and checking for consistency.

Chapter 4

SYNTAX ANALYSIS AND CODE GENERATION

In this chapter we will discuss the last two major components of the translator which generates the simulators. These are the syntax analyzer and code generator.

Let us first look at the implementation of syntax analysis, which checks the input to see if it is legal. If there is an error in the input, various error recovery schemes are tried so that parsing can continue [Bau,Tin,Wil]. If the error recovery attempt fails, the whole program is aborted and no simulator is generated.

The parser that does this syntax analysis is driven by a state table (STATE!TABLE). STATE!TABLE was generated from the state diagram in Fig 4.1a. States 17 and 18 are missing from the state diagram, having been deleted by standard state reduction methods. Input to the state table consists of the previous state of the table and the tokenised input (Fig 4.1b).

A scanner reads the input and breaks it into meaningful groups of characters (names, symbols, numbers, etc.). The scanner then classifies each group of characters with a classification technique, with the final result being a tokenised input.

Selecting the next state and doing error recovery is the function of a procedure called STATE!MACHINE. The output from STATE!MACHINE is either the next state or the number zero, indicating an error from which recovery is not possible (terminal error).

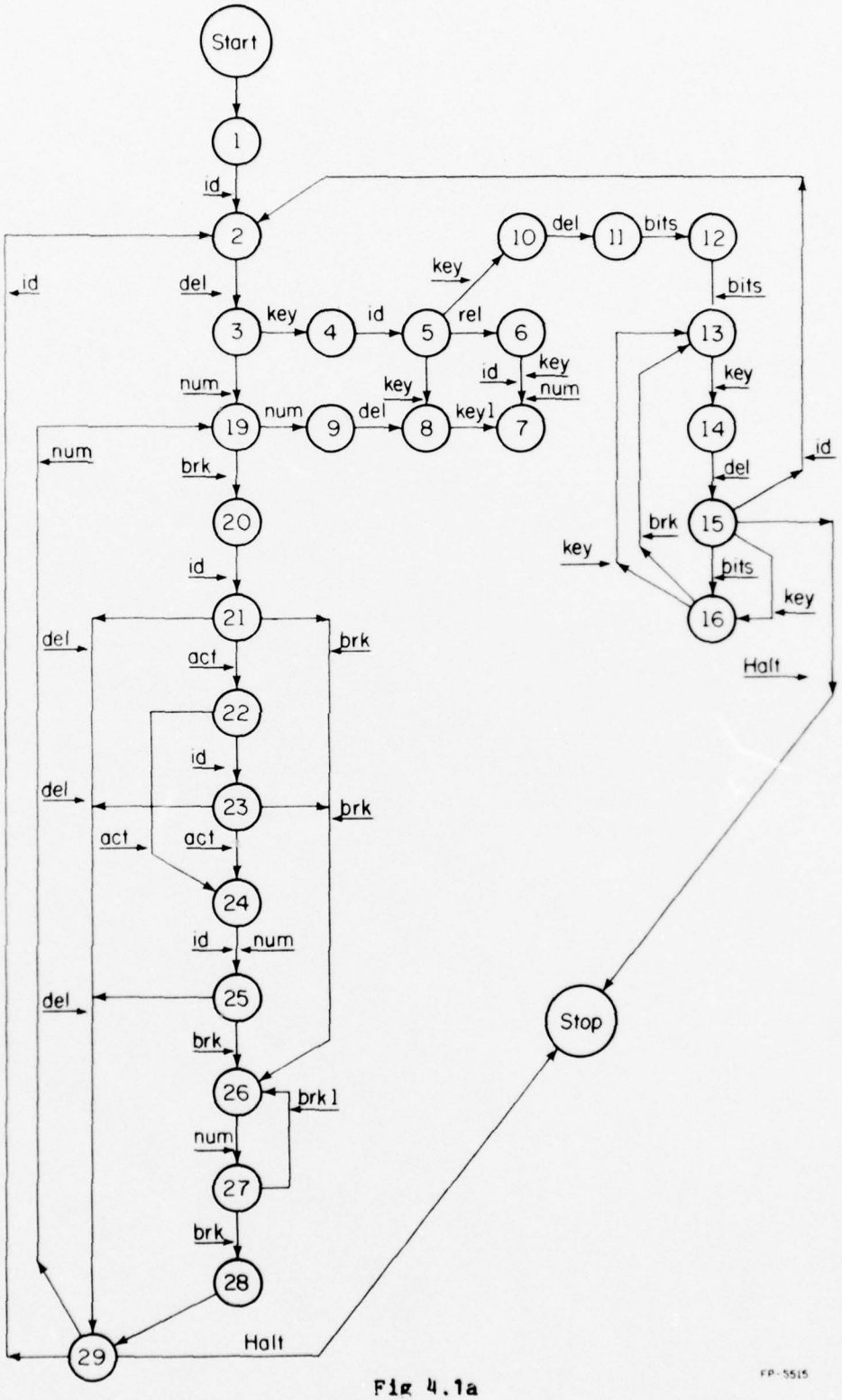


Fig 4.1a

State Diagram

TOKENS		INPUT
id	=	Identifiers.
key	=	Keywords except "DO".
key1	=	The keyword "DO".
num	=	Numbers.
bits	=	Binary numbers (only 0´s and 1´s).
act	=	Operators (+, -, AND, &, SHL).
rel	=	Binary relations (<, >, >=, <>).
brk	=	Symbols "(" or ")".
brk1	=	Symbol ",".
del	=	End of the line (cr or ";").
Halt	=	End of the input.

Fig 4.1b

Tokens and their corresponding input characters

A variety of error recovery schemes is tried depending on what type of error occurred. Some of our methods may appear draconian, but this simplifies the error recovery. One method is to flush and ignore the current line and then to create a dummy line of input that the parser will accept, thereby allowing the parser to continue. However, with the straightforward grammar and syntax structure of the DL, the use of such harsh methods should be infrequent.

The parser itself is very straightforward. It consists of a large CASE statement whose index variable is the next state of the state table. This CASE statement is within a large loop which loops until either the end of the input is reached or a terminal error occurs. The action of the parser can most easily be followed by tracing through its actions with the state diagram (Fig 4.1). The parser's documentation will explain the actions at each state.

The parser works on a block by block basis, where each block is translated into a SAIL procedure. At the entry to each block, the parser calls a routine(END!OF!BLOCK) that checks the ordering of the previous block, serializes the block's basic statements and outputs them. Then another routine (NEW!BLOCK) will initialize some key variables in preparation for parsing the current block .

At the entry to each block, the parser first determines the type of block (PROC, WPROC, or DPROC). The WPROC and DPROC blocks require some extra code (the block's preamble) to be generated at the beginning of the procedure. In the case of the WPROC, a WHILE loop is created with the predicate and a counter which controls the

loop, to prevent infinite looping.

The parser's code is generally straightforward with most of it given to checking input legality. The only place the coding is awkward is in the handling of the statements for the DPROC. They are:

binary-number DOES Proc-call

binary-number DOES COMPLETE

NONE DOES Proc-call

NONE DOES COMPLETE.

If there is more than one NONE statement, we ignore all but the first. Generally code is generated on a statement-by-statement basis, but code is not generated for a NONE statement at this point in parsing; instead, we wait until the entire block is written out and then generate some closing code called the DECODE!POSTAMBLE. This postamble generates the appropriate code for the NONE statement. If a NONE statement does not occur in the DPROC, code will be generated to catch values that are not covered by the DPROC's statements, and an error message will be displayed.

Two different types of code are generated. One type of code is tailored for the DPROC statements and the other type of code is for the register transfer and proc-call statements in the WPROC and PROC blocks.

The code generated for the DPROC statements was basically covered in the preceding paragraphs, so we will not discuss the DPROC statements any further. We will also ignore code generation for the proc-call since it is very simple. Instead, we will look at the more interesting and difficult problems of generating tight code for register transfer statements. First, some terminology is needed. The element on the left hand side of the assignment will be called the "object" or the "object of the assignment". Both operands will be called operands with no distinction made between them. The term "result" is used to indicate the value generated by an operation, and a temporary result is a temporary that has been generated with the value of the "result". "Vector" will be used to denote any member of Vector-class.

Before code can be generated for a statement, several checks must be made. First, are there sufficient operands for the operator and can the operation cause an overflow? Second, are the operands and object subfields? If so, will we need to generate intermediate results? Third, are the data paths of sufficient width for the operation? (A 7 bit result cannot be put in a 4 bit object field.)

There are four routines associated with code generation: MACRO!NAME, OPN!CHK, WRT!REG!TRF, and OBJ!GEN.

MACRO!NAME is very simple; it is passed the operator and returns the name of the macro that performs the operation, the number of operands required by the operator, and a flag indicating if the operator can cause an overflow.

OPN!CHK is called from the parser; its input is a register transfer operand. OPN!CHK makes some decisions concerning the operand, which could be a subfield, a vector, or a register. If the operand is a subfield, the subfield must be extracted; if the operand is a vector a function for handling this class of elements must be generated. The output of OPN!CHK is a string that is used directly as an operand for the macro generated by MACRO!NAME. The string will belong to one of three categories. If the operand is a subfield, a temporary is assigned to the extracted field and the name of the temporary is returned. If the operand is a vector, a function that handles this type of data element is returned. (There are at present three groups of functions. Each group has a function that retrieves an element and inserts an element. MEMORY, STACK and QUEUE each have a group (2) of functions associated with it.) If the operand is a register, the operand is returned with one modification: the operand is ANDed with a "ones" mask to extract the proper size bit field.

WRT!REG!TRF has the job of first checking to see if the data path requirements are met and secondly to call OBJ!GEN. WRT!REG!TRF will take the operator name and concatenate it with the operands to form a macro which is passed to OBJ!GEN. It will also pass the hash number of the object, a switch indicating if the operation can cause an overflow(OVER!FLOW), and a switch indicating if the right hand side expression is "simple" or not(SIMPLE!OP).

OBJ!GEN does the same as OPN!CHK except that it works on the object and not the operands. OBJ!GEN will generate code similar to that of OPN!CHK, depending on whether the object is a subfield, vector, or a register. The SIMPLE!OP and OVERFLOW switches are used to minimize the amount of code generated. If SIMPLE!OP is on, no "temporary result" is generated. If the OVERFLOW switch is on, then when generating code to set status conditions, the overflow checking code will be generated.

Looking at the process as a whole, we see that OPN!CHK will receive each operand and return either a string with the modified operand, the name of a temporary where the operand's current value now resides, or a function for properly accessing the element. MACRO!NAME will take the operator and return its associated macro and some switches. WRT!REG!TRF will concatenate the macro name and the operands to form a macro representing the right hand side of the statement. This will be passed to OBJ!GEN, which will generate the final code for the register transfer statement and, in some cases, code to set status conditions.

This concludes the discussion of syntax analysis and code generation for the individual blocks. Next we consider in what order we output the code for execution. Since we have a serial machine, all the parallelism within a block must be serialized for execution without violating the ordering that was declared. This brings forth the second problem: insuring that the statements form a partial ordering.

Using the routine TOPOLOGICAL!SORT, we test if the statements for the current block form a partial ordering. This routine returns in MAPPER the statement numbers in a total ordering without violating the original partial ordering. If a partial ordering does not exist, code for this block is not outputted, and the routine returns a "false" value which is ANDed with previous GOOD!BLOCK!SW's, thus any incorrect block will cause GOOD!BLOCK!SW to be false. If the block is syntactically correct, WRITE!OUT!BLOCK is called to output the code and the checkpoints. This will continue until the end of the input is encountered.

Next we check the inter-block connections, INTER!BLOCK, to see if any direct cycles exist. TOPOLOGICAL!SORT is also used to check this ordering. If direct cycles exist, PGM!SW is set to false. If direct cycles do not exist, we write out the last lines of code for the simulator. We must check if the simulator has been improperly generated. This is done by ANDing GOOD!BLOCK!SW and PGM!SW together. If the result is false, a nonrecoverable error occurred and the simulator is destroyed. If the result is true, a simulator has been generated. The SAIL compiler compiles the simulator and executes it. At the end of the execution, we have the simulation's execution time output and the simulator, which the user can now run at any time.

Chapter 5

DESIGN EXAMPLE

In this chapter we will look at a design example which was developed and tested using the translator.

The design is a simple computer called the Single Instruction Machine (SIM). As SIM's name implies it executes a single subtract and test instruction, SUBTST A, B, P , see top of Fig 5.1. The machine has been thoroughly described in two previous articles by Mudge. The only changes made have been the addition of a data structure and the insertion of a checkpoint. This is not the only DS possible; for example, it could have easily been made 16 bits wide rather than 12 bits wide as chosen here.

Fig 5.1 is the program which SIM will execute as a test program. The program calculates the Fibonacci sequence until the WPROC's built in counter reaches its predetermined value and stops looping (see page 24). The program is loaded into memory via the INITIALIZE statement; other key variables PC and RUN are initialized in the same way. Fig 5.2 is the input to the translator. Fig 5.3 is the simulator that was generated by the translator, and Fig 5.4 is the simulator's output(i.e. the simulation), which is generated by the CHECKPOINT. When the PC is equal to 2 and 5 the DR is a calculated Fibonacci number. At the other locations DR has intermediate results. The PC is written out after the PC has been incremented therefore this does not conflict with the test program which shows the Fibonnaci number being calculated at locations 1 and 4.

SIM
A single intruction machine

SUBTST A, B, P
1. A<- C[A] - C[B]
2. IF C[A] = 0 THEN PC<-C[P]

```
LOC
0 LOOP: SUBTST ZERO,B          ! ZERO<--B
1       SUBTST A,ZERO          ! A<-B - ZERO
2       SUBTST ZERO,ZERO,NEXT  ! ZERO<-ZERO-ZERO
3 NEXT: SUBTST ZERO,A          ! ZERO<--A
4       SUBTST B,ZERO          ! B<-B-ZERO
5       SUBTST ZERO,ZERO,LOOP  ! ZERO<-ZERO-ZERO
6       ZERO=0   !  ZERO IS A CONSTANT
7       A=1      !  INITIAL VALUE
8       B=1      !  INITIAL VALUE
```

Fig 5.1
Program to calculate the Fibonnaci sequence

SIM
DATA STRUCTURE and CONTROL STRUCTURE

```
   INITIAL 104,118,870,103,134,102,0,1,1 ;
   MEMORY M(16) (12)
   MEM!ADDRER MAR(12) FOR M


   INITIAL ON;
   REGISTER RUN(1)
   REGISTER INT(1)
   REGISTER IR(12)
   SUBFIELD IRP(4) TO IR(8)
   SUBFIELD IRA(4) TO IR(4)
   SUBFIELD IRB(4) TO IR(0)

   INITIAL 0;
   REGISTER PC(4)
   REGISTER DR(12)
   REGISTER DR1(12)
   REGISTER DR2(12)
   REGISTER A(12)
   REGISTER IPA(12)
   REGISTER IP(12)

   BLOCK SIM;
   BLOCK DECI
   BLOCK FETCH
   BLOCK EXEC
   BLOCK INTR
   BLOCK DEC
   BLOCK BR

   END!DATA

SIM
WHILE TRUE  DO
 1) DECI
 2) FETCH  (1)
 3) EXEC (2)

DECI
DECODE INT AS
    1 DOES  INTR
    0 DOES  COMPLETE
```

(Fig 5.2)

```
INTR
1) MAR<-IPA
2) DR<-IP
3) M<-DR (1,2)

FETCH
1) MAR<-PC
2) DR<-M (1)
3) PC<-INC PC (1)

EXEC
1) IR<-DR
12) DEC (10,11)
11) M<-DR (8,9)
2) MAR<-IRA (1)
3) DR<-M (2)
4) DR1<-DR (3)
5) A<-DR (3)
6) MAR<-IRB (3)
7) DR<-M (4,5,6)
8) DR<-A -DR (7)
9) MAR<-IRA (7)
10) DR2<-DR (8)
0) CHKPT DR,PC (12)

DEC
DECODE DR AS
  0 DOES  BR
NONE DOES  COMPLETE

BR
1) DR<-PC
2) PC<-IRP (1)
3) MAR<-PC (2)
4) DR<-M (3)
   END
```

Fig 5.2
Design example for SIM
input to the translator

```
BEGIN "MAIN"
REQUIRE "MACROS.SAI" SOURCE!FILE;
REQUIRE "DCL.SAI" SOURCE!FILE;
PRELOAD!WITH  104,118,870,103,134,102,0,1,1 ;

     INTEGER ARRAY M[0:16];
     INTEGER MAR;
     INTEGER RUN;
     INTEGER INT;
     INTEGER IR;
     INTEGER PC;
     INTEGER DR;
     INTEGER DR1;
     INTEGER DR2;
     INTEGER A;
     INTEGER IPA;
     INTEGER IP;
     FORWARD PROCEDURE SIM;
     FORWARD PROCEDURE DECI;
     FORWARD PROCEDURE FETCH;
     FORWARD PROCEDURE EXEC;
     FORWARD PROCEDURE INTR;
     FORWARD PROCEDURE DEC;
     FORWARD PROCEDURE BR;

REQUIRE "FUNC.SAI" SOURCE!FILE;


PROCEDURE SIM;
BEGIN "SIM"
PROCESS!NAME<-"SIM";
WPROC!PRE(TRUE  );
DECI;
FETCH;
EXEC;
WPROC!POSTAMBLE
RETURN;
END "SIM";

PROCEDURE DECI;
BEGIN "DECI"
PROCESS!NAME<-"DECI";
DECODE!PREAMBLE(INT,0,1);
DECODE!ST(1,INTR,0);
DECODE!ST(0,INTR,1);
DECODE!POSTAMBLE(NULL);
RETURN;
END "DECI";
```

(Fig 5.3)

```
   PROCEDURE INTR;
   BEGIN "INTR"
   PROCESS!NAME<-"INTR";
   MAR<-IPA;
   DR<-IP;
   WRITE!MEM(M,"M",16,4095,MAR,4095,0,DR);
   RETURN;
   END "INTR";

   PROCEDURE FETCH;
   BEGIN "FETCH"
   PROCESS!NAME<-"FETCH";
   MAR<-PC;
   DR<-READ!MEM(M,"M",16,4095,MAR,4095,0);
   PC<-INCR(<PC>);
   RETURN;
   END "FETCH";

   PROCEDURE EXEC;
   BEGIN "EXEC"
   PROCESS!NAME<-"EXEC";
   IR<-DR;
   EXTRACTOR(1,IR,4,15);
   MAR<-TMP[1];
   DR<-READ!MEM(M,"M",16,4095,MAR,4095,0);
   DR1<-DR;
   A<-DR;
   EXTRACTOR(1,IR,0,15);
   MAR<-TMP[1];
   DR<-READ!MEM(M,"M",16,4095,MAR,4095,0);
   DR<-SUBTRACT(<A>,<DR>);
   EXTRACTOR(1,IR,4,15);
   MAR<-TMP[1];
   DR2<-DR;
   WRITE!MEM(M,"M",16,4095,MAR,4095,0,DR);
   DEC;

   OUT(CHANNEL!WRT,CRLF&" CHKPT FOR STATMENT   12 IN PROCESS
EXEC");
   CHKPTS("DR",DR,0,0,4095,-1,0);
   CHKPTS("PC",PC,0,0,15,-1,0);

   RETURN;
   END "EXEC";

   PROCEDURE DEC;
   BEGIN "DEC"
   PROCESS!NAME<-"DEC";
   DECODE!PREAMBLE(DR,0,4095);
   DECODE!ST(0,BR,0);
   DECODE!POSTAMBLE(NULL);
   RETURN;
   END "DEC";
```

(Fig 5.3)

```
PROCEDURE BR;
BEGIN "BR"
PROCESS!NAME<-"BR";
DR<-PC;
EXTRACTOR(1,IR,8,15);
PC<-TMP[1];
MAR<-PC;
DR<-READ!MEM(M,"M",16,4095,MAR,4095,0);
RETURN;
END "BR";

! INITIALIZATION OF VARIABLES;

RUN<- ON;
PC<- 0;

MAX!TIMES!THRU!LOOP<-30;
CHANNEL!WRT<-OPENFILE("FILE","W");
SIM;
CLOSE(CHANNEL!WRT);
END "MAIN";
```

Fig 5.3
The simulator for SIM based on the given
data structure and control structure

```
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4095
PC=1
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=2
PC=2
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=103
PC=3
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4094
PC=4
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=3
PC=5
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=104
PC=0
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4093
PC=1
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=5
PC=2
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=103
PC=3
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4091
PC=4
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=8
PC=5
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=104
PC=0
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4088
PC=1
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=13
PC=2
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=103
PC=3
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4083
PC=4
```

(Fig 5.4)

```
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=21
PC=5
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=104
PC=0
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4075
PC=1
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=34
PC=2
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=103
PC=3
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4062
PC=4
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=55
PC=5
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=104
PC=0
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4041
PC=1
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=89
PC=2
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=103
PC=3
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=4007
PC=4
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=144
PC=5
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=104
PC=0
 CHKPT FOR STATMENT   12 IN PROCESS EXEC
DR=3952
PC=1
```

Fig 5.4
Output from the simulator generated by the checkpoint

Chapter 6

CONCLUSION

In this concluding chapter, we will discuss the methods used to build the translator, and we will look at some areas where the translator can be improved or expanded.

The translator has two major components; the data base component and syntax/code generation component. Each of these components is encapsulated in its own procedure, thus preventing interaction except through external variables. All of the routines exchange information through procedure parameters or external variables. Modularization is one of the key features of the translator. This modularity allows us to change routines in one module without affecting other routines, unlike the case of closely intertwined routines. Because of this feature, the translator can be expanded, changed, repaired, and maintained easily. This is very important since any system that is frequently used will be undergoing constant change and maintenance as new bugs are found and repaired.

The translator itself makes heavy use of macros internally as well as for code generation purposes. Macros were used to parameterize the translator so changes in constants would require changes only in the macros. This reduced the chances of error. We also used macros to replace sections of code that were frequently used, to implement sections of code that involved shared variables, and to replace similar lines of code throughout the translator. This improved the readability and understandability of the program.

By these methods, the translator can be easily maintained and improved in the future.

As we built this translator, many features were encountered that we felt would be necessary in the system if it were to be used. Many of the features would be difficult to implement and could consume a great deal of time, while others are replacements of current routines which, in hindsight, could have been improved on, and some weren't needed to get a basic translator going.

Two of the easier changes possible concern the data base and setting of translator constants externally. The routines dealing with the data base should be replaced or improved. The present routines aren't very flexible nor are they very tolerant of input error. One of the problems is the proliferation of keywords in the declaration statements ("the more keywords the greater the chance of error"). An example is the keywords prefixed with "subfld"; this prefix can be eliminated and the syntax can be allowed instead to declare the item to be a subfield.

At present, if certain bounds constants are to be changed(SYMBOL!TABLE size, maximum number of basic statements in a block), the values assigned to macro names must be changed in the translator. This is clumsy and inefficient. Routines could be written to allow the user to set these constants from the console. There are two approaches: either query the user from the translator or read the constants off a command line. This latter approach is used by the SAIL compiler to set switches and acquire constants.

Other areas are more difficult to implement and much more thought will be involved in designing them. One area is very important, and the other two areas would be useful and necessary if the system is to be frequently used.

The most important area is that of determinism. Two possible alternative solutions to the problem have been suggested and some "hooks" have been left in the translator on which to hang one of the possible solutions (the "set" solution). This problem should receive priority, since a large scale system cannot be checked for determinism by "eyeballing" the design. This was the method used to check the determinism of SIM in Chapter 5.

At some time someone will want to test his simulator under the influence of interrupts. This will involve designing routines to generate interrupts(i.e. random number generators), and some means of defining the desired interrupt structure, and translating it for use by the simulator. Thus, a designer could specify, perhaps in the data base, the interrupt system he wants (such as the PDP/8's interrupt system or a more complex PDP/11 type interrupt structure), as well as how the interrupts will appear.

At present if the user has designed and created a simulator for machine "X" and wishes to add, delete, or change the CHECKPOINTs, he must change the translator's input and retranslate the entire design. The simulator would be the same with CHECKPOINTs in different locations. It would be more convenient if the user could specify CHECKPOINTs from the terminal before the simulator was run or during simulation like a debug package. Routines for doing this

will be quite large since ID!LIST and SYMBOL!TABLE must be kept along with tables containing the location of each basic statement, and the location of all the current CHECKPOINTs.

These routines cover the more important areas for growth in the translator, and the user is sure to have many more ideas. The present version of the translator is basic. It can handle small designs quite well, and with some of the additional routines mentioned, it will be a very powerful tool for designing large digital systems.

REFERENCES

Bar72 Barbacci, M.R., Bell,C.G. and Newell,A., "ISP: A Language to Describe Instruction Sets and Other Register Transfer Systems", Proceedings IEEE Computer Conference, COMPCON 72, pp. 219-222, September 1972.

Bar75 Barbacci, M.R., "A Comparsion of Register Transfer Languages for Describing Computers and Digital Systems", IEEE TC, Vol. c-24, No. 2 pp. 137-150, February 1975.

Bau Bauer, F.L. and Eickel, J. (eds), Compiler Construction, Springer-Verlag, Berlin, 1974.

Chu72a Chu, Y., "Introducing the Computer Design Language", Proceeding IEEE Computer Conference, COMPCON 72, pp. 215-218, September 1972.

Chu72b Chu, Y., "Significance of the SYMBOL Computer System", Proceeding IEEE Computer Conference, COMPCON 72, pp. 33-36, September 1972.

Chu77 Chu,Y., "Architecture of a Hardware Data Interpreter", 4th Annual Symposium on Computer Architecture, pp. 1-9, March 1977.

Cof Coffman, E.G., and Denning, P.J., Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 35-44.

Dul Duley, J.P. and Dietmeyer, D.L., "A Digital System Design Language (DDL)", IEEE TC, Vol. c-17, No. 9, pp. 850-861, September 1968.

Joh Johnson, D.B., "Finding all the Elementary Circuits of a Directed Graph", SIAM Journal on Computing, Vol. 4, No 1, pp. 77-84, March 1975.

Knu Knuth, D.E., Fundamental Algorithims, Vol. 1, Addison-Wesley, Reading, Mass., 1969, pp. 258-265.

Met Metze, G. and Seshu, A., "A Proposal for a Computer Compiler", Proceeding Spring Joint Computer Conference, pp. 253-263, 1966.

Mud75 Mudge, T., "Specifying a Design Language for Digital Systems", Proceeding 13th Annual Allerton Conference on Circuit and System Theory, pp. 905-915, October 1975.

Mud77 Mudge, T., and Metze, G., A Design Language for Modular Asynchronous Control Structures, CSL R-759, University of Illinois, February 1977.

Pet Peterson, J.B., On High Level Digital System Design, CSL R-653, University of Illinois, July 1974.

Rei Reiser, J.F., (ed), SAIL, Computer Science Department Report No. STAN-CS-76-574, Stanford University, August 1976.

Smi Smith, W.R., "System Supervision Algorithims for the SYMBOL Computer", Proceeding IEEE Computer Conference, pp. 21-25, September 1972.

Tin Tindall, M.H., An Interactive Compile-Time Diagnostic System, UIUCDCS-R-75-748, University of Illinois, pp. 10-20, October 1975.

Wil Wilcox, T.R., "Compiler Construction", class notes Spring 1976.